Cuestionario Teórico Clases 6, 7 y 8

1. ***Defina y diferencie programa concurrente, programa distribuido y programa paralelo.***

Definiciones

- **Programa concurrente:** Un programa concurrente contiene dos o más procesos que trabajan juntos para realizar una tarea. Cada proceso es un programa secuencial; es decir, una secuencia de sentencias que se ejecutan una tras otra. Mientras que un programa secuencial tiene un solo hilo de control, un programa concurrente tiene múltiples hilos de control. Los procesos de un programa concurrente trabajan juntos comunicándose entre sí.
- **Programa distribuido:** Un programa distribuido es un programa concurrente en el que los procesos se ejecutan en diferentes procesadores que no comparten memoria.
- **Programa paralelo:** Un programa paralelo es un programa concurrente en el que los procesos se ejecutan en paralelo en varios procesadores. El objetivo de la computación paralela es resolver un problema dado más rápidamente o, de forma equivalente, resolver un problema más grande en la misma cantidad de tiempo.

Diferencias clave

- **Arquitectura:** Los programas concurrentes pueden ejecutarse en un único procesador o en varios procesadores. Los programas distribuidos se ejecutan en varios procesadores que no comparten memoria. Los programas paralelos se ejecutan en varios procesadores que pueden o no compartir memoria.
- **Comunicación:** Los programas concurrentes pueden comunicarse mediante variables compartidas o paso de mensajes. Los programas distribuidos se comunican mediante paso de mensajes. Los programas paralelos pueden comunicarse mediante variables compartidas o paso de mensajes.
- **Objetivo:** El objetivo de los programas concurrentes es gestionar múltiples actividades independientes. El objetivo de los programas distribuidos es resolver problemas que implican datos distribuidos o mejorar la fiabilidad. El objetivo de los programas paralelos es mejorar el rendimiento resolviendo problemas más rápidamente o resolviendo problemas más grandes en la misma cantidad de tiempo.

2. ***Marque al menos 2 similitudes y 2 diferencias entre los pasajes de mensajes sincrónicos y asincrónicos.***

Similitudes

- **Ambos se utilizan para la comunicación y la sincronización entre procesos.** Tanto el paso de mensajes sincrónico como el asincrónico permiten a los procesos comunicarse entre sí enviando y recibiendo mensajes. También pueden utilizarse

para sincronizar procesos haciendo que un proceso espere a recibir un mensaje de otro proceso antes de continuar.
- **Ambos pueden utilizarse en sistemas de memoria distribuida.** Tanto el paso de mensajes sincrónico como el asincrónico son apropiados para su uso en sistemas donde los procesos se ejecutan en diferentes procesadores que no comparten memoria.

Diferencias

- **Bloqueo:** En el paso de mensajes sincrónico, la operación de envío es bloqueante, lo que significa que el proceso emisor se bloquea hasta que el proceso receptor recibe el mensaje. En el paso de mensajes asincrónico, la operación de envío no es bloqueante.
- **Complejidad de la implementación:** El paso de mensajes asincrónico suele ser más complejo de implementar que el paso de mensajes sincrónico. Esto se debe a que el sistema debe proporcionar un mecanismo para almacenar los mensajes que se han enviado pero que aún no se han recibido.

3. ***Analice qué tipo de mecanismos de pasaje de mensajes son más adecuados para resolver problemas de tipo Cliente/Servidor, Pares que interactúan, Filtros y, Productores y Consumidores. Justifique claramente su respuesta.***

Cliente/Servidor

- Para problemas de tipo Cliente/Servidor, **tanto el Pasaje de Mensajes Asincrónico (PMA) como el Sincrónico (PMS) pueden ser adecuados**, pero existen ventajas y desventajas en cada caso:
    - **PMA:** Permite mayor concurrencia, ya que el cliente puede seguir trabajando después de enviar una solicitud al servidor sin esperar la respuesta. Esto es útil en situaciones donde el cliente no necesita la respuesta inmediatamente. La implementación de servidores con PMA se asemeja a la de los monitores, donde las solicitudes se manejan como procedimientos remotos. La dualidad entre monitores y PMA permite traducir la estructura de un monitor a un servidor basado en mensajes.
    - **PMS:** Es más simple de implementar que el PMA, ya que no requiere un mecanismo para almacenar mensajes en búfer. Sin embargo, puede limitar la concurrencia, ya que el cliente se bloquea hasta que el servidor responde.
- **RPC (Llamado a Procedimiento Remoto) y Rendezvous son alternativas más adecuadas para problemas Cliente/Servidor** que los mecanismos básicos de paso de mensajes. Estos mecanismos combinan la interfaz procedural de los monitores con el paso de mensajes implícitos, simplificando la programación y ocultando la complejidad de la comunicación entre procesos.

Pares que interactúan

- Para problemas de tipo Pares que interactúan, donde los procesos necesitan intercambiar información y sincronizarse entre sí, **el PMS puede ser más adecuado**:
    - **PMS:** Permite una sincronización más directa entre los pares, ya que la operación de envío bloquea al emisor hasta que el receptor recibe el mensaje. Esto simplifica la lógica de sincronización en comparación con el PMA.
    - **PMA:** Aunque se puede utilizar, requiere mecanismos adicionales para garantizar la sincronización y evitar condiciones de carrera.
- **La comunicación guardada, presente en lenguajes como CSP, ofrece una solución eficiente para la interacción entre pares.** Esta técnica permite a los procesos esperar selectivamente mensajes de diferentes fuentes, lo que facilita la programación de interacciones complejas.

Filtros

- Para problemas de tipo Filtros, donde los procesos se conectan en una cadena para procesar datos secuencialmente, **el PMA es generalmente más adecuado**:
    - **PMA:** Permite mayor concurrencia, ya que cada filtro puede trabajar en su parte del flujo de datos sin esperar a que los demás terminen. Esto se debe a que los canales actúan como colas de mensajes, permitiendo que los filtros envíen datos sin bloquearse.
    - **PMS:** Puede generar bloqueos y limitar la concurrencia, ya que cada filtro debe esperar a que el siguiente esté listo para recibir antes de poder enviar.

Productores y Consumidores

- Para problemas de tipo Productores y Consumidores, donde un proceso genera datos que son consumidos por otro, **tanto PMA como PMS pueden ser adecuados, dependiendo de la necesidad de concurrencia**:
    - **PMA:** Ofrece mayor concurrencia, ya que el productor puede seguir generando datos sin esperar a que el consumidor los procese. Esto se logra utilizando un canal como búfer entre el productor y el consumidor.
    - **PMS:** Si bien reduce la concurrencia, puede ser útil para controlar el flujo de datos entre el productor y el consumidor, asegurando que el productor no genere datos más rápido de lo que el consumidor puede procesar.

4. ***Indique por qué puede considerarse que existe una dualidad entre los mecanismos de monitores y pasaje de mensajes. Ejemplifique.***

- Se puede considerar que existe una dualidad entre los mecanismos de monitores y pasaje de mensajes porque cada uno puede simular al otro. Esto significa que cualquier problema de sincronización que se pueda resolver con monitores también se puede resolver con pasaje de mensajes, y viceversa.

5. ***¿En qué consiste la comunicación guardada (introducida por CSP) y cuál es su utilidad? Describa cómo es la ejecución de sentencias de alternativa e iteración que contienen comunicaciones guardadas***.

- La comunicación guardada, introducida por CSP, permite a los procesos comunicarse de forma no determinística y selectiva. En lugar de bloquearse en una operación de entrada o salida, un proceso puede esperar a que se cumplan ciertas condiciones o a que se reciban mensajes específicos.

Utilidad:

- **Flexibilidad en la comunicación:** Permite a los procesos comunicarse con múltiples procesos sin un orden predeterminado, adaptándose a la disponibilidad de los demás procesos.
- **Prevención de bloqueos:** Evita que un proceso se bloquee indefinidamente esperando un mensaje que nunca llegará.
- **Manejo eficiente de la concurrencia:** Permite que los procesos continúen su ejecución mientras esperan mensajes, aumentando el grado de concurrencia.

Sintaxis:

- Las sentencias de comunicación guardada en CSP tienen la forma: **B; C → S;**

  Donde:

  - **B:** Condición booleana opcional. Si se omite, se asume como true.
  - **C:** Sentencia de comunicación (entrada o salida).
  - **S:** Conjunto de sentencias que se ejecutan si la guarda tiene éxito.

Evaluación de una guarda:

- Una guarda puede tener tres resultados:
  - **Éxito:** La condición booleana B es verdadera (o no existe) y la comunicación C se puede realizar sin demora (el otro proceso está listo para comunicarse).
  - **Fallo:** La condición booleana B es falsa, independientemente de la sentencia de comunicación.
  - **Bloqueo:** La condición booleana B es verdadera (o no existe), pero la comunicación C no se puede realizar sin demora (el otro proceso aún no está listo para comunicarse).

Ejecución de Sentencias con Comunicación Guardada

Sentencias de Alternativa (if guardada):

1. **Evaluación de guardas:** Se evalúan todas las guardas.
2. **Selección de guarda:**
   - Si todas las guardas fallan, la sentencia if termina sin efecto.
   - Si al menos una guarda tiene éxito, se elige una de ellas de forma no determinística.

- Si algunas guardas se bloquean, se espera hasta que alguna tenga éxito.

3. **Ejecución:** Se ejecuta la sentencia de comunicación C de la guarda elegida y luego el conjunto de sentencias S asociado a esa guarda.

Ejemplo:

```
Process P1
{   int valor1 = 1, valor2;
    if  P2 ! (valor1) → P2 ? (valor2);
    □ P2 ? (valor2) → P2 ! (valor1);
    fi
}

Process P2
{   int valor1 , valor2 = 2;
    if  P1 ! (valor2) → P1 ? (valor1);
    □ P1 ? (valor1) → P1 ! (valor2);
    fi
}
```

Sentencias de Iteración (do guardada):

1. **Evaluación de guardas:** Se evalúan todas las guardas.
2. **Selección de guarda:**
   - Si todas las guardas fallan, la sentencia do termina.
   - Si al menos una guarda tiene éxito, se elige una de ellas de forma no determinística.
   - Si algunas guardas se bloquean, se espera hasta que alguna tenga éxito.
3. **Ejecución:** Se ejecuta la sentencia de comunicación C de la guarda elegida y luego el conjunto de sentencias S asociado a esa guarda.
4. **Repetición:** Se repiten los pasos 1 a 3 hasta que todas las guardas fallen.

Ejemplo:

```
Process Copiar
{  char buffer[80];
   int front = 0, rear = 0, cantidad = 0;
   do cantidad <80; Oeste?(buffer[rear]) →  cantidad = cantidad + 1;
                                            rear = (rear + 1) MOD 80;
   □  cantidad >0; Este!(buffer[front]) → cantidad := cantidad - 1;
                                          front := (front + 1) MOD 80;
   od
}
```

6. ***Modifique la solución con mensajes sincrónicos de la Criba de Eratóstenes para encontrar los números primos detallada en teoría de modo que los procesos no terminen en deadlock.***

- La solución de la Criba de Eratóstenes con PMS, como se describe en la fuente, presenta un problema de deadlock al utilizar comunicación síncrona. Al finalizar la generación de primos, los procesos Criba[i] (para i > 1) quedan bloqueados indefinidamente esperando un mensaje de su predecesor (Criba[i-1]), que ya no enviará más números.

  Para evitar este deadlock, se puede implementar una estrategia de terminación utilizando un valor especial o "centinela" que marque el final del flujo de datos.

Modificación del Algoritmo

1. **Introducir un centinela:** Se define un valor especial (por ejemplo, 0) como centinela para indicar el fin del stream de números primos.
2. **Enviar el centinela al final:** El proceso Criba, después de enviar todos los números impares, enviará el centinela a Criba.
3. **Propagar el centinela:** Cada proceso Criba[i], al recibir el centinela:
    - Imprimirá su valor de p (el primo que está filtrando).
    - Enviará el centinela a su sucesor (Criba[i+1]), si existe.
    - Terminará su ejecución.

Código Modificado

```
Process Criba {
  """
  Proceso principal que inicializa la criba y envía los números impares
  (posibles primos) al primer proceso de la cadena. Envía un centinela (0)
  al final para indicar el término de los candidatos.
  """

  int p = 2;  // Inicializar el primer primo en 2

  // Enviar solo números impares como candidatos a Criba[2]
  for [i = 3 to n by 2] Criba[2] ! (i);

  // Enviar centinela para finalizar la cadena
  Criba[2] ! (0);
}

Process Criba[i = 2 to L] {
  """
  Cada instancia del proceso Criba[i] filtra números múltiplos del primo p
  recibido del proceso anterior (Criba[i-1]). Solo transmite los números que
  no son múltiplos de p al siguiente proceso en la cadena. Al recibir el centinela,
  imprime el valor de p (como un número primo confirmado) y propaga el centinela
  para finalizar la cadena.
  """

  int p, proximo;  // Declarar variables para el primo y el próximo candidato

  // Recibir el primer valor de p, que representa el nuevo primo
  Criba[i-1] ? (p);

  do
    Criba[i-1] ? (proximo) →
      if (proximo == 0) {
        """
        Si se recibe el centinela (0), se imprime el valor de p como un número
        primo confirmado y se envía el centinela al siguiente proceso (si existe)
        para propagar el fin de la secuencia.
        """

        print(p);  // Imprimir el primo confirmado
        if (i < L) Criba[i+1] ! (0);  // Propagar el centinela
      } else {
        if (proximo MOD p <> 0) {
          """
          Si proximo no es múltiplo de p, entonces no es divisible por el
          primo actual, y puede ser enviado al siguiente proceso para
          continuar la verificación.
          """

          if (i < L) Criba[i+1] ! (proximo);  // Enviar al siguiente proceso
        }
      }
  od
}
```

7. ***Explique brevemente los 7 paradigmas de interacción entre procesos en programación distribuida vistos en teoría. En cada caso ejemplifique, indique qué tipo de comunicación por mensajes es más conveniente y qué arquitectura de hardware se ajusta mejor. Justifique sus respuestas***.

Master/Worker (Bolsa de Tareas)

- **Descripción:** Un proceso master distribuye tareas a varios procesos worker. Los workers ejecutan las tareas y envían los resultados al master.
- **Ejemplo:** Un programa para procesar imágenes que divide la imagen en varias partes y asigna cada parte a un worker.
- **Comunicación:** Asincrónica (PMA) es generalmente más conveniente, ya que el master puede distribuir tareas sin esperar a que los workers terminen.
- **Arquitectura:** Se adapta a sistemas distribuidos, especialmente a clusters de computadoras.

Heartbeat

- **Descripción:** Los procesos intercambian mensajes periódicamente para indicar que están activos y para compartir información de estado.
- **Ejemplo:** Un sistema de monitoreo de red que utiliza heartbeat para detectar fallos en los nodos.
- **Comunicación:** Asincrónica (PMA) es generalmente más conveniente, ya que los mensajes heartbeat no requieren una respuesta inmediata.
- **Arquitectura:** Se adapta a sistemas distribuidos, especialmente a aquellos que requieren alta disponibilidad.

Pipeline

- **Descripción:** Los datos fluyen a través de una secuencia de procesos, donde cada proceso realiza una etapa de procesamiento.
- **Ejemplo:** Un compilador que divide la compilación en varias etapas (análisis léxico, análisis sintáctico, generación de código) y asigna cada etapa a un proceso.
- **Comunicación:** Sincrónica (PMS) o asincrónica (PMA), dependiendo del algoritmo y la necesidad de sincronización.
- **Arquitectura:** Se adapta a diversas arquitecturas, desde un único procesador hasta sistemas distribuidos.

Probe/Echo

- **Descripción:** Un proceso envía mensajes de prueba (probes) a sus sucesores en un grafo o árbol. Los sucesores responden con mensajes de eco (echoes).
- **Ejemplo:** Un algoritmo para determinar la topología de una red.
- **Comunicación:** Asincrónica (PMA) es generalmente más conveniente, ya que los probes y echoes no requieren una sincronización estricta.

- **Arquitectura:** Se adapta a sistemas distribuidos, especialmente a aquellos con una estructura de grafo o árbol.

Broadcast

- **Descripción:** En este paradigma, un proceso envía un mensaje idéntico a todos los demás procesos en el sistema. Este método es útil para diseminar información rápidamente.
- **Ejemplo:** Un proceso que anuncia un cambio de estado a todos los demás.
- **Tipo de Comunicación:** Se puede implementar con mensajes asincrónicos, aunque no es una operación atómica.
- **Arquitectura:** Redes que soportan la primitiva de broadcast, como muchas LAN.
- **Consideraciones:**
    - **Ordenamiento:** Los mensajes enviados por diferentes procesos pueden llegar en distinto orden a los receptores.
    - **Fiabilidad:** Se requiere de mecanismos para garantizar que todos los procesos reciban el mensaje.
    - **Aplicaciones:** Semáforos distribuidos, detección de terminación distribuida.

Token Passing

- **Descripción:** Un token, que puede ser un mensaje especial o un permiso, circula entre los procesos. Solo el proceso que posee el token puede realizar ciertas acciones.
- **Ejemplo:** Controlar el acceso a un recurso compartido en un sistema distribuido.
- **Tipo de Comunicación:** Generalmente se implementa con mensajes asincrónicos.
- **Arquitectura:** Se puede implementar en diferentes arquitecturas, pero es más común en sistemas distribuidos con topologías específicas como anillos.
- **Consideraciones:**
    - **Tolerancia a Fallos:** Se necesitan mecanismos para manejar la pérdida del token.
    - **Complejidad:** El diseño de algoritmos de token passing puede ser complejo.
    - **Aplicaciones:** Exclusión mutua distribuida, detección de terminación distribuida.

Servidores Replicados

- **Descripción:** Se crean múltiples instancias de un servidor para mejorar la disponibilidad, la tolerancia a fallos y la escalabilidad. Cada servidor maneja una réplica del recurso.
- **Ejemplo:** Un sistema de archivos distribuido con copias de archivos en varios nodos.
- **Tipo de Comunicación:** Puede usar tanto mensajes sincrónicos como asincrónicos, dependiendo de la implementación.
- **Arquitectura:** Sistemas distribuidos, especialmente aquellos donde la alta disponibilidad es crucial.
- **Consideraciones:**

    - **Consistencia:** Se necesita un mecanismo para mantener la consistencia entre las réplicas.
    - **Complejidad de la Gestión:** La administración de servidores replicados puede ser compleja.
    - **Aplicaciones:** Sistemas de archivos distribuidos, bases de datos distribuidas, servidores web.

Justificación de las arquitecturas de hardware:

- Las arquitecturas distribuidas son las más adecuadas para la mayoría de estos paradigmas, ya que permiten que los procesos se ejecuten en diferentes máquinas, lo que aumenta la escalabilidad y la tolerancia a fallos.
- Las arquitecturas de memoria compartida también pueden ser adecuadas para algunos paradigmas, como Productor/Consumidor y Pipeline, si la comunicación entre procesos es frecuente y la cantidad de datos transferidos es pequeña.

Comunicación sincrónica vs. asincrónica:

- La elección entre comunicación sincrónica y asincrónica depende de la necesidad de sincronización entre los procesos y del impacto en el rendimiento.
- La comunicación sincrónica es más sencilla de programar, pero puede bloquear los procesos y reducir el grado de concurrencia.
- La comunicación asincrónica es más compleja de programar, pero puede aumentar el grado de concurrencia y el rendimiento.

8. Describa el paradigma “bag of tasks”. ¿Cuáles son las principales ventajas del mismo?

El Paradigma "Bag of Tasks"

El paradigma "bag of tasks" (bolsa de tareas) es una técnica popular en la computación paralela para resolver problemas que pueden descomponerse en tareas independientes. En este modelo, se mantiene una colección de tareas, la "bolsa", a la que varios procesos "trabajadores" pueden acceder.

- Cada trabajador toma una tarea de la bolsa, la ejecuta y, opcionalmente, puede generar nuevas tareas que se agregan a la bolsa.
- Este proceso continúa hasta que no queden tareas en la bolsa y todos los trabajadores estén inactivos.
- La implementación del paradigma "bag of tasks" se puede realizar mediante variables compartidas o con paso de mensajes.

Ventajas del Paradigma "Bag of Tasks"

Este paradigma ofrece varias ventajas, incluyendo:

- **Simplicidad*:*** Es fácil de entender e implementar.
- **Escalabilidad*:*** Se puede usar con cualquier número de procesadores simplemente ajustando la cantidad de workers.
- **Balanceo de Carga*:*** Facilita el balanceo de carga ya que los workers toman tareas a medida que las necesitan, asegurando que la carga de trabajo se distribuya de forma relativamente uniforme.
- **Flexibilidad*:*** Se puede aplicar a una variedad de problemas, incluyendo aquellos que requieren la generación dinámica de tareas.

9. ***Suponga n^2 procesos organizados en forma de grilla cuadrada. Cada proceso puede comunicarse sólo con los vecinos izquierdo, derecho, de arriba y de abajo (los procesos de las esquinas tienen solo 2 vecinos, y los otros en los bordes de la grilla tienen 3 vecinos). Cada proceso tiene inicialmente un valor local v.***
    a. ***Escriba un algoritmo heartbeat que calcule el máximo y el mínimo de los n2 valores. Al terminar el programa, cada proceso debe conocer ambos valores. (Nota: no es necesario que el algoritmo esté optimizado).***
    b. ***Analice la solución desde el punto de vista del número de mensajes.***
    c. ***¿Puede realizar alguna mejora para reducir el número de mensajes?***
    d. ***Modifique la solución de a) para el caso en que los procesos pueden comunicarse también con sus vecinos en las diagonales.***

```
Proceso P[i, j] { // i, j son las coordenadas del proceso en la grilla
  // Inicialización
  int v = valor_local;
  int minimo_local = v;
  int maximo_local = v;

  // Iteraciones
  para (int ronda = 0; ronda < n; ronda++) { // n rondas

    // Fase de envío
    enviar(minimo_local, maximo_local) a vecino_arriba;
    enviar(minimo_local, maximo_local) a vecino_abajo;
    enviar(minimo_local, maximo_local) a vecino_izquierda;
    enviar(minimo_local, maximo_local) a vecino_derecha;

    // Fase de recepción
    recibir(minimo_vecino, maximo_vecino) de vecino_arriba;
    recibir(minimo_vecino, maximo_vecino) de vecino_abajo;
    recibir(minimo_vecino, maximo_vecino) de vecino_izquierda;
    recibir(minimo_vecino, maximo_vecino) de vecino_derecha;

    // Fase de actualización
    minimo_local = min(minimo_local, minimo_vecino_arriba,
                       minimo_vecino_abajo, minimo_vecino_izquierda,
                       minimo_vecino_derecha);
    maximo_local = max(maximo_local, maximo_vecino_arriba,
                       maximo_vecino_abajo, maximo_vecino_izquierda,
                       maximo_vecino_derecha);
  }

  // Finalización
  // minimo_local y maximo_local ahora contienen el mínimo y máximo global
}
```

Explicación:

- El algoritmo utiliza un enfoque iterativo donde cada proceso se comunica con sus vecinos en cada ronda.
- En cada ronda, los procesos envían sus valores minimo_local y maximo_local a sus vecinos y reciben los valores correspondientes de ellos.
- Luego, actualizan sus propios valores tomando el mínimo y el máximo de los valores recibidos y su propio valor actual.
- Después de n rondas, se garantiza que la información se ha propagado por toda la grilla, y cada proceso tendrá el mínimo y máximo global.

Nota:

- El algoritmo asume que cada proceso conoce las direcciones de sus vecinos.

10. Marque similitudes y diferencias entre los mecanismos RPC y Rendezvous. Ejemplifique para la resolución de un problema a su elección.

Tanto **RPC (Llamada a Procedimiento Remoto)** como **Rendezvous** son mecanismos de comunicación y sincronización entre procesos en sistemas distribuidos, especialmente adecuados para aplicaciones cliente/servidor. Ambos combinan una interfaz similar a la de los monitores, con operaciones exportadas invocables mediante llamadas externas (CALL), con el uso de mensajes sincrónicos. Esto significa que el proceso que realiza la llamada se bloquea hasta que la operación llamada se complete y se devuelvan los resultados.

Similitudes:

- **Interfaz similar a un monitor:** Ambos mecanismos permiten definir módulos con operaciones (o procedimientos) exportadas que pueden ser invocadas por procesos en otros módulos.
- **Comunicación sincrónica:** En ambos casos, la llamada a una operación es bloqueante. El proceso llamador se suspende hasta que el proceso servidor complete la operación y devuelva los resultados.
- **Orientación a Cliente/Servidor:** RPC y Rendezvous son especialmente útiles para implementar interacciones cliente/servidor, donde un servidor ofrece servicios a múltiples clientes.

Diferencias:

- **Manejo de la invocación:** La principal diferencia radica en cómo se maneja la invocación de una operación:
    - **RPC:** Se crea un nuevo proceso (o se utiliza uno de un pool preexistente) para manejar cada llamada. El proceso servidor ejecuta el procedimiento correspondiente a la operación, envía los resultados al llamador y finaliza.
    - **Rendezvous:** La invocación se realiza mediante un "encuentro" (rendezvous) con un proceso servidor ya existente. Este proceso utiliza una sentencia de entrada (in) para esperar una invocación, procesarla y devolver los resultados. La atención de las operaciones se realiza de forma secuencial, no concurrente.
- **Sincronización:**
    - **RPC:** El mecanismo en sí mismo sólo proporciona comunicación. La sincronización entre los procesos dentro de un módulo debe programarse explícitamente utilizando semáforos, monitores u otros mecanismos.
    - **Rendezvous:** Combina comunicación y sincronización. La sentencia de entrada (in) del proceso servidor se encarga de la sincronización, bloqueando al servidor hasta que llegue una invocación y al cliente hasta que se complete la operación.

11. ***Considere el problema de lectores/escritores. Desarrolle un proceso servidor para implementar el acceso a la base de datos, y muestre las interfaces de los lectores y escritores con el servidor. Los procesos deben interactuar: a) con mensajes asincrónicos; b) con mensajes sincrónicos; c) con RPC; d) con Rendezvous.***

Mensajes Asincrónicos

```
chan solicitud(int tipoProceso, int idProceso); // tipoProceso: 0=lector, 1=escritor
chan respuesta[n](int estado); // estado: 0=denegado, 1=concedido

process Servidor {
  int numLectores = 0;
  boolean escritorActivo = false;
  queue<int> lectoresEspera;

  while (true) {
    int tipoProceso, idProceso;
    receive solicitud(tipoProceso, idProceso);

    if (tipoProceso == 0) { // lector
      if (escritorActivo) {
        lectoresEspera.push(idProceso);
      } else {
        numLectores++;
        send respuesta[idProceso](1); // concedido
      }
    } else { // escritor
      if (numLectores > 0 || escritorActivo) {
        send respuesta[idProceso](0); // denegado
      } else {
        escritorActivo = true;
        send respuesta[idProceso](1); // concedido
      }
    }
  }
}

process Lector[i = 0 to n-1] {
  send solicitud(0, i);
  int estado;
  receive respuesta[i](estado);
  if (estado == 1) {
    // leer la base de datos
    // protocolo de liberación
  } else {
    // manejar el acceso denegado
  }
}

process Escritor[i = 0 to m-1] {
  send solicitud(1, i);
  int estado;
  receive respuesta[i](estado);
  if (estado == 1) {
    // escribir en la base de datos
    // protocolo de liberación
  } else {
    // manejar el acceso denegado
  }
}
```

Protocolo de Liberación (Lector):

- El lector envía un mensaje asincrónico al servidor indicando que ha terminado.
- El servidor decrementa numLectores.
- Si numLectores es 0 y hay escritores esperando, el servidor concede el acceso al primer escritor en la cola.

Protocolo de Liberación (Escritor):

- El escritor envía un mensaje asincrónico al servidor indicando que ha terminado.
- El servidor establece escritorActivo en falso.
- Si hay lectores esperando, el servidor concede el acceso a todos los lectores en la cola. Si no hay lectores esperando, concede el acceso al primer escritor en la cola.

Mensajes Sincrónicos

La implementación con mensajes sincrónicos es similar a la de mensajes asincrónicos, con la diferencia de que las primitivas enviar y recibir bloquean al proceso hasta que se complete la comunicación. Esto simplifica la lógica de liberación, ya que el proceso que libera el recurso puede hacerlo directamente sin necesidad de un protocolo asincrónico.

Observación: Las fuentes proporcionan ejemplos de cómo usar mensajes sincrónicos para la comunicación, pero no se proporciona un algoritmo específico para el problema de lectores/escritores.

RPC

```
Módulo Servidor:
modulo Servidor {
  // Procedimientos exportados
  procedure solicitarAcceso(tipoProceso: entero, idProceso: entero) : entero;
  procedure liberarAcceso(tipoProceso: entero);
}

body Servidor {
  // Variables compartidas
  int numLectores = 0;
  boolean escritorActivo = falso;
  semaforo mutex;
  semaforo okLeer;
  semaforo okEscribir;

  // Inicialización de semáforos
  init(mutex, 1);
  init(okLeer, 0);
  init(okEscribir, 0);

  // Implementación de solicitarAcceso
  procedure solicitarAcceso(tipoProceso: entero, idProceso: entero) : entero {
    P(mutex);
    si (tipoProceso == 0) { // lector
      si (escritorActivo) {
        V(mutex);
        P(okLeer);
        P(mutex);
      }
      numLectores++;
      V(okLeer);
    } sino { // escritor
      si (numLectores > 0 o escritorActivo) {
        V(mutex);
        P(okEscribir);
        P(mutex);
      }
      escritorActivo = verdadero;
    }
    V(mutex);
    devolver 1; // acceso concedido
  }
```

```
  // Implementación de liberarAcceso
  procedure liberarAcceso(tipoProceso: entero) {
    P(mutex);
    si (tipoProceso == 0) { // lector
      numLectores--;
      si (numLectores == 0 y hay escritores esperando) {
        V(okEscribir);
      }
    } sino { // escritor
      escritorActivo = falso;
      si (hay lectores esperando) {
        V(okLeer);
      } sino si (hay escritores esperando) {
        V(okEscribir);
      }
    }
    V(mutex);
  }
}

Proceso Lector:
proceso Lector[i = 0 a n-1] {
  estado = Servidor.solicitarAcceso(0, i);
  si (estado == 1) {
    leer la base de datos;
    Servidor.liberarAcceso(0);
  } sino {
    // manejar el acceso denegado
  }
}

Proceso Escritor:
proceso Escritor[i = 0 a m-1] {
  estado = Servidor.solicitarAcceso(1, i);
  si (estado == 1) {
    escribir en la base de datos;
    Servidor.liberarAcceso(1);
  } sino {
    // manejar el acceso denegado
  }
}
```

Rendezvous

```
-- Módulo Servidor
task type Servidor is
  entry solicitarAcceso(tipoProceso : Integer);
  entry liberarAcceso(tipoProceso : Integer);
end Servidor;

task body Servidor is
  numLectores : Integer := 0;
  escritorActivo : Boolean := False;
begin
  loop
    select
      when tipoProceso = 0 and not escritorActivo =>
        accept solicitarAcceso(tipoProceso : Integer) do
          numLectores := numLectores + 1;
        end solicitarAcceso;
    or
      when tipoProceso = 1 and numLectores = 0 and not escritorActivo =>
        accept solicitarAcceso(tipoProceso : Integer) do
          escritorActivo := True;
        end solicitarAcceso;
    or
      accept liberarAcceso(tipoProceso : Integer) do
        if tipoProceso = 0 then
          numLectores := numLectores - 1;
          if numLectores = 0 and hayEscritoresEsperando then
            -- desbloquear a un escritor
          end if;
        elsif tipoProceso = 1 then
          escritorActivo := False;
          if hayLectoresEsperando then
            -- desbloquear a todos los lectores
          elsif hayEscritoresEsperando then
            -- desbloquear a un escritor
          end if;
        end if;
      end liberarAcceso;
    end select;
  end loop;
end Servidor;

task type Lector;
lectores : array (1 .. m) of Lector;

-- Proceso Lector
task body Lector is
begin
  Servidor.solicitarAcceso(0);
  -- leer la base de datos
  Servidor.liberarAcceso(0);
end Lector;

task type Escritor;
escritores : array (1 .. m) of Escritor;

-- Proceso Escritor
task body Escritor is
begin
  Servidor.solicitarAcceso(1);
  -- escribir en la base de datos
  Servidor.liberarAcceso(1);
end Escritor;
```

12. ***Describa sintéticamente las características de sincronización y comunicación de ADA.***

ADA utiliza el **rendezvous** como su principal mecanismo de sincronización y comunicación, ideal para programar aplicaciones cliente/servidor.

- Los puntos de entrada a una tarea se llaman "**entrys**".
- La tarea que declara un entry utiliza la primitiva "**accept**" para servir llamadas a ese entry.
- "**accept**" detiene la tarea hasta que haya una invocación, copia los parámetros, ejecuta las sentencias y devuelve los resultados al llamador.

Tipos de Entrys

- Simples y familias de entrys con parámetros de entrada (IN), salida (OUT) o entrada/salida (IN OUT).

Tipos de Llamadas a Entrys

- **Llamada a Entry:** detiene al llamador hasta que la operación termine.
- **Llamada a Entry Condicional:** se ejecuta solo si la llamada se puede realizar inmediatamente.
- **Llamada a Entry Temporal:** se ejecuta solo si la llamada se puede realizar dentro de un tiempo límite.

Comunicación Guardada

- La sentencia "**select**" permite al servidor elegir entre diferentes alternativas de accept.
- Cada línea del select es una "**alternativa**" que puede incluir una cláusula "**when**" para especificar una condición.
- Se puede usar "**else**", "**delay**" o "**terminate**" en las alternativas.